LEARN C++
Skill up with our free tutorials

2.12 — Header guards

ALEX[1] AUGUST 29, 2023

The duplicate definition problem

In lesson 2.7 -- Forward declarations and definitions (https://www.learncpp.com/cpp-tutorial/forward-declarations/)[2], we noted that a variable or function identifier can only have one definition (the one definition rule). Thus, a program that defines a variable identifier more than once will cause a compile error:

```
int main()
{
    int x; // this is a definition for variable x
    int x; // compile error: duplicate definition

    return 0;
}
```

Similarly, programs that define a function more than once will also cause a compile error:

```
#include <iostream>

int foo() // this is a definition for function foo
{
    return 5;
}

int foo() // compile error: duplicate definition
{
    return 5;
}

int main()
{
    std::cout << foo();
    return 0;
}
```

While these programs are easy to fix (remove the duplicate definition), with header files, it’s quite easy to end up in a situation where a definition in a header file gets included more than once. This can happen when a header file #includes another header file (which is common).

Author’s note

In upcoming examples, we’ll define some functions inside header files. You generally shouldn’t do this.

We are doing so here because it is the most effective way to demonstrate some concepts using functionality we've already covered.

Consider the following academic example:

square.h:

```cpp
int getSquareSides()
{
    return 4;
}
```

wave.h:

```cpp
#include "square.h"
```

main.cpp:

```cpp
#include "square.h"
#include "wave.h"

int main()
{
    return 0;
}
```

This seemingly innocent looking program won't compile! Here's what's happening. First, *main.cpp* #includes *square.h*, which copies the definition for function *getSquareSides* into *main.cpp*. Then *main.cpp* #includes *wave.h*, which #includes *square.h* itself. This copies contents of *square.h* (including the definition for function *getSquareSides*) into *wave.h*, which then gets copied into *main.cpp*.

Thus, after resolving all of the #includes, *main.cpp* ends up looking like this:

```cpp
int getSquareSides()  // from square.h
{
    return 4;
}

int getSquareSides() // from wave.h (via square.h)
{
    return 4;
}

int main()
{
    return 0;
}
```

Duplicate definitions and a compile error. Each file, individually, is fine. However, because *main.cpp* ends up #including the content of *square.h* twice, we've run into problems. If *wave.h* needs *getSquareSides()*, and *main.cpp* needs both *wave.h* and *square.h*, how would you resolve this issue?

Header guards

The good news is that we can avoid the above problem via a mechanism called a **header guard** (also called an **include guard**). Header guards are conditional compilation directives that take the following form:

```
#ifndef SOME_UNIQUE_NAME_HERE
#define SOME_UNIQUE_NAME_HERE

// your declarations (and certain types of definitions) here

#endif
```

When this header is #included, the preprocessor checks whether *SOME_UNIQUE_NAME_HERE* has been previously defined. If this is the first time we're including the header, *SOME_UNIQUE_NAME_HERE* will not have been defined. Consequently, it #defines *SOME_UNIQUE_NAME_HERE* and includes the contents of the file. If the header is included again into the same file, *SOME_UNIQUE_NAME_HERE* will already have been defined from the first time the contents of the header were included, and the contents of the header will be ignored (thanks to the #ifndef).

All of your header files should have header guards on them. *SOME_UNIQUE_NAME_HERE* can be any name you want, but by convention is set to the full filename of the header file, typed in all caps, using underscores for spaces or punctuation. For example, *square.h* would have the header guard:

square.h:

```
#ifndef SQUARE_H
#define SQUARE_H

int getSquareSides()
{
    return 4;
}

#endif
```

Even the standard library headers use header guards. If you were to take a look at the iostream header file from Visual Studio, you would see:

```
#ifndef _IOSTREAM_
#define _IOSTREAM_

// content here

#endif
```

For advanced readers

In large programs, it's possible to have two separate header files (included from different directories) that end up having the same filename (e.g. directoryA\config.h and directoryB\config.h). If only the filename is used for the include guard (e.g. CONFIG_H), these two files may end up using the same guard name. If that happens, any file that includes (directly or indirectly) both config.h files will not receive the contents of the include file to be included second. This will probably cause a compilation error.

Because of this possibility for guard name conflicts, many developers recommend using a more complex/unique name in your header guards. Some good suggestions are a naming convention of PROJECT_PATH_FILE_H, FILE_LARGE-RANDOM-NUMBER_H, or FILE_CREATION-DATE_H.

Updating our previous example with header guards

Let's return to the *square.h* example, using the *square.h* with header guards. For good form, we'll also add header guards to *wave.h*.

square.h

```
#ifndef SQUARE_H
#define SQUARE_H

int getSquareSides()
{
    return 4;
}

#endif
```

wave.h:

```
#ifndef WAVE_H
#define WAVE_H

#include "square.h"

#endif
```

main.cpp:

```
#include "square.h"
#include "wave.h"

int main()
{
    return 0;
}
```

After the preprocessor resolves all of the #include directives, this program looks like this:

main.cpp:

```
// Square.h included from main.cpp
#ifndef SQUARE_H // square.h included from main.cpp
#define SQUARE_H // SQUARE_H gets defined here

// and all this content gets included
int getSquareSides()
{
    return 4;
}

#endif // SQUARE_H

#ifndef WAVE_H // wave.h included from main.cpp
#define WAVE_H
#ifndef SQUARE_H // square.h included from wave.h, SQUARE_H is already defined from above
#define SQUARE_H // so none of this content gets included

int getSquareSides()
{
    return 4;
}

#endif // SQUARE_H
#endif // WAVE_H

int main()
{
    return 0;
}
```

Let's look at how this evaluates.

First, the preprocessor evaluates `#ifndef SQUARE_H`. `SQUARE_H` has not been defined yet, so the code from the `#ifndef` to the subsequent `#endif` is included for compilation. This code defines `SQUARE_H`, and has the definition for the `getSquareSides` function.

Later, the next `#ifndef SQUARE_H` is evaluated. This time, `SQUARE_H` is defined (because it got defined above), so the code from the `#ifndef` to the subsequent `#endif` is excluded from compilation.

Header guards prevent duplicate inclusions because the first time a guard is encountered, the guard macro isn't defined, so the guarded content is included. Past that point, the guard macro is defined, so any subsequent copies of the guarded content are excluded.

Header guards do not prevent a header from being included once into different code files

Note that the goal of header guards is to prevent a code file from receiving more than one copy of a guarded header. By design, header guards do *not* prevent a given header file from being included (once) into separate code files. This can also cause unexpected problems. Consider:

square.h:

```
#ifndef SQUARE_H
#define SQUARE_H

int getSquareSides()
{
    return 4;
}

int getSquarePerimeter(int sideLength); // forward declaration for getSquarePerimeter

#endif
```

square.cpp:

```
#include "square.h"  // square.h is included once here

int getSquarePerimeter(int sideLength)
{
    return sideLength * getSquareSides();
}
```

main.cpp:

```
#include "square.h" // square.h is also included once here
#include <iostream>

int main()
{
    std::cout << "a square has " << getSquareSides() << " sides\n";
    std::cout << "a square of length 5 has perimeter length " << getSquarePerimeter(5) << '\n';

    return 0;
}
```

Note that *square.h* is included from both *main.cpp* and *square.cpp*. This means the contents of *square.h* will be included once into *square.cpp* and once into *main.cpp*.

Let's examine why this happens in more detail. When *square.h* is included from *square.cpp*, *SQUARE_H* is defined until the end of *square.cpp*. This define prevents *square.h* from being included into *square.cpp* a second time (which is the point of header guards). However, once *square.cpp* is finished, *SQUARE_H* is no longer considered defined. This means that when the preprocessor runs on *main.cpp*, *SQUARE_H* is not initially defined in *main.cpp*.

The end result is that both *square.cpp* and *main.cpp* get a copy of the definition of *getSquareSides*. This program will compile, but the linker will complain about your program having multiple definitions for identifier *getSquareSides*!

The best way to work around this issue is simply to put the function definition in one of the .cpp files so that the header just contains a forward declaration:

square.h:

```
#ifndef SQUARE_H
#define SQUARE_H

int getSquareSides(); // forward declaration for getSquareSides
int getSquarePerimeter(int sideLength); // forward declaration for
getSquarePerimeter

#endif
```

square.cpp:

```
#include "square.h"

int getSquareSides() // actual definition for getSquareSides
{
    return 4;
}

int getSquarePerimeter(int sideLength)
{
    return sideLength * getSquareSides();
}
```

main.cpp:

```
#include "square.h" // square.h is also included once here
#include <iostream>

int main()
{
    std::cout << "a square has " << getSquareSides() << " sides\n";
    std::cout << "a square of length 5 has perimeter length " <<
getSquarePerimeter(5) << '\n';

    return 0;
}
```

Now when the program is compiled, function *getSquareSides* will have just one definition (via *square.cpp*), so the linker is happy. File *main.cpp* is able to call this function (even though it lives in *square.cpp*) because it includes *square.h*, which has a forward declaration for the function (the linker will connect the call to *getSquareSides* from *main.cpp* to the definition of *getSquareSides* in *square.cpp*).

Can't we just avoid definitions in header files?

We've generally told you not to include function definitions in your headers. So you may be wondering why you should include header guards if they protect you from something you shouldn't do.

There are quite a few cases we'll show you in the future where it's necessary to put non-function definitions in a header file. For example, C++ will let you create your own types. These custom types are typically defined in header files, so the type definitions can be propagated out to the code files that need to use them. Without a header guard, a code file could end up with multiple (identical) copies of a given type definition, which the compiler will flag as an error.

So even though it's not strictly necessary to have header guards at this point in the tutorial series, we're establishing good habits now, so you don't have to unlearn bad habits later.

#pragma once

Modern compilers support a simpler, alternate form of header guards using the `#pragma` preprocessor directive:

```
#pragma once

// your code here
```

`#pragma once` serves the same purpose as header guards: to avoid a header file from being included multiple times. With traditional header guards, the developer is responsible for guarding the header (by using preprocessor directives `#ifndef`, `#define`, and `#endif`). With `#pragma once`, we're requesting that the compiler guard the header. How exactly it does this is an implementation-specific detail.

For advanced readers

There is one known case where `#pragma once` will typically fail. If a header file is copied so that it exists in multiple places on the file system, if somehow both copies of the header get included, header guards will successfully de-dupe the identical headers, but `#pragma once` won't (because the compiler won't realize they are actually identical content).

For most projects, `#pragma once` works fine, and many developers now prefer it because it is easier and less error-prone. Many IDEs will also auto-include `#pragma once` at the top of a new header file generated through the IDE.

Warning

The `#pragma` directive was designed for compiler implementers to use for whatever purposes they desire. As such, which pragmas are supported and what meaning those pragmas have is completely implementation-specific. With the exception of `#pragma once`, do not expect a pragma that works on one compiler to be supported by another.

Because `#pragma once` is not defined by the C++ standard, it is possible that some compilers may not implement it. For this reason, some development houses (such as Google) recommend using traditional header guards. In this tutorial series, we will favor header guards, as they are the most conventional way to guard headers. However, support for `#pragma once` is fairly ubiquitous at this point, and if you wish to use `#pragma once` instead, that is generally accepted in modern C++.

Summary

Header guards are designed to ensure that the contents of a given header file are not copied more than once into any single file, in order to prevent duplicate definitions.

Duplicate *declarations* are fine -- but even if your header file is composed of all declarations (no definitions) it's still a best practice to include header guards.

Note that header guards do *not* prevent the contents of a header file from being copied (once) into separate project files. This is a good thing, because we often need to reference the contents of a given header from different project files.

Quiz time

Question #1

Add header guards to this header file:

add.h:

```
int add(int x, int y);
```

Show Solution (javascript:void(0))3

Next lesson
2.13 How to design your first programs

4

Back to table of contents

5

Previous lesson
2.11 Header files

6

7

B U URL INLINE CODE C++ CODE BLOCK HELP!

Leave a comment...

Name*

Email*

Notify me about replies:

POST COMMENT

Find a mistake? Leave a comment above!

Avatars from https://gravatar.com/[10] are connected to your provided email address.

566 COMMENTS Newest

Baker Alshaif
April 19, 2024 2:53 pm

What if only a declarations in a header file collide with another header file? Don't header guards just guard the header if an exact copy of the header is used again?

0 Reply

Alex Author
Reply to Baker Alshaif[11] April 20, 2024 3:10 pm

> What if only a declarations in a header file collide with another header file?

Then you'll likely get a compilation error. This is why namespaces are a good thing.

0 Reply

Baker
Reply to Alex[12] April 20, 2024 3:55 pm

No I mean what if only one declaration collides, not all.

0 Reply

Alex Author
Reply to Baker[13] April 24, 2024 1:35 pm

Same thing. If any declaration collides, then you should get a compilation error.

0 Reply

Nicolas
April 2, 2024 7:49 am

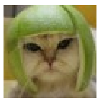
Is it ok if i use both header guards AND #pragma once on header files?

```
#pragma once
#ifndef SUBTRACT_H
#define SUBTRACT_H

int subtract(int x, int y);

#endif
```

0 Reply



Alex Author
Reply to Nicolas 14 April 2, 2024 2:11 pm

Yes.

2 Reply

Almaz
March 14, 2024 11:15 pm

Thank you so much for this tutorial, it's really beautiful. The authors, you have a lot of respect for continuing to develop this textbook, introducing something new, changing if something is outdated, and responding to comments. I LOVE YOU and thank you

8 Reply

Misfit Devil
March 9, 2024 9:19 am

On the CONFIG_H example, you said there's a possibility of name clash.
Is there anything wrong in setting a random sequence for the file? For example, in the `add.h` exercise:

```
#ifndef ac91b1bb-3e74-4940-8e6f-902fc0f5a60b
#define ac91b1bb-3e74-4940-8e6f-902fc0f5a60b

int add(int x, int y);

#endif
```

Or is this considered bad practice?

0 Reply

Alex Author
Reply to Misfit Devil [15] March 11, 2024 5:31 pm

Using a string-formatted GUID is fine.

0 Reply

Mykolas
March 2, 2024 8:26 am

"The best way to work around this issue is simply to put the function definition in one of the .cpp files so that the header just contains a forward declaration:". But you have also told us that avoiding definitions in header files is not always possible (e.g. when defining a custom type). So if I need that header file in multiple of my programs files, that would violate the one definition rule, would'nt it?

0 Reply

Alex Author
Reply to Mykolas [16] March 4, 2024 6:33 pm

No, because the definitions we typically put in header files (types, templates, and inline functions) are all exempt from the one definition rule.

1 Reply

Paster
February 29, 2024 5:07 pm

```cpp
//add.cpp

int add(int x , int y)
{
    return x + y;
}

//add.h
#pragma once

int add(int x,int y);

//main.cpp

#include "add.h"
#include <iostream>

int main()
{
   std::cout << add(2,3) << '\n';
}
```

Is it a good pratice?

0 Reply

Alex Author

Reply to Paster 17 March 4, 2024 11:41 am

Looks good to me.

0 Reply

Bob

February 29, 2024 12:55 am

Thank you for the amazing tutorial! I have a question :
In the previous lesson Introduction to the preprocessor: https://www.learncpp.com/cpp-tutorial/introduction-to-the-preprocessor/ At "The scope of #defines" section, it says "Once the preprocessor has finished, all defined identifiers from that file are discarded. This means that directives are only valid from the point of definition to the end of the file in which they are defined. Directives defined in one code file do not have impact on other code files in the same project". And it also says "Directives are resolved before compilation."
How is #define SOME_UNIQUE_NAME_HERE effective in another file?

0 Reply

Alex Author

Reply to Bob 18 March 4, 2024 11:31 am

Directives defined in a file are valid within the translation unit (which includes any #included content).

Therefore, `#define SOME_UNIQUE_NAME_HERE` in one translation unit is not effective in another translation unit (unless it also appears there). The goal of a header guard is to prevent duplicate inclusion within a single translation unit, not within different translation units.

0 Reply

Aaron
January 10, 2024 5:46 am

Hello friends. Please tell me the mistake I've made:

I have copied the code for main.cpp, square.h, and wave.h from section "Updating our previous example with header guards"
and receive these errors:

Severity Code Description Project File Line Suppression State Details
Error LNK2005 "int __cdecl getSquareSides(void)" (?getSquareSides@@YAHXZ) already defined in main.obj Project2 C:\Users\aaron\source\repos\Project2\square.obj 1

Severity Code Description Project File Line Suppression State Details
Error LNK2005 "int __cdecl getSquareSides(void)" (?getSquareSides@@YAHXZ) already defined in main.obj Project2 C:\Users\aaron\source\repos\Project2\wave.obj 1

Severity Code Description Project File Line Suppression State Details
Error LNK1169 one or more multiply defined symbols found Project2 C:\Users\aaron\source\repos\Project2\x64\Debug\Project2.exe 1

I appreciate your assistance.

Last edited 3 months ago by Aaron

0 Reply

Alex Author
Reply to Aaron 19 January 11, 2024 1:19 pm

Somehow you're getting duplicate definitions of `getSquareSides()`. You either have a typo in your header guards, or you are compiling in a file containing a duplicate definition somehow.

0 Reply

mark
December 4, 2023 5:18 am

I am confused in the sense that in the previous lesson we foward-declared the function in add.h then defined it in the add.cpp(of course with inclusion of "add.h") . why is that we cannot do that here even with the introduction of header guards? correct me if i am wrong alex- as for me i think that after the header guards we just foward declare the function in square.h and then declare it in square.cpp and then finally include "square.h" in main.cpp

please help me ...i haven't come this far only to get stopped at this point

0 Reply

Alex Author
Reply to mark 20 December 6, 2023 4:04 pm

That's exactly what you should do.

That said, header guards aren't needed to guard against forward declarations, as multiple identical forward declarations are allowed.

0 Reply

Krishnakumar
November 16, 2023 3:42 am

>With the exception of #pragma once, do not expect a pragma that works on one compiler to be supported by another.

This is not fully true. The OpenMP and OpenACC standards used in high performance computing rely heavily on #pragma directives. There are standardisation committees for that, along with officially maintained lists of supported compiler implementations and the version of the OpenMP standards they comply with.

0 Reply

Krishnakumar
November 16, 2023 2:57 am

```
#ifndef _IOSTREAM_
#define _IOSTREAM_
```

The `iostream` file on my system has the following header guard.

```
#ifndef _GLIBCXX_IOSTREAM
#define _GLIBCXX_IOSTREAM 1
```

0 Reply

Krishnakumar
October 11, 2023 4:25 am

Apparently, as per official microsoft docs here: https://learn.microsoft.com/en-us/cpp/preprocessor/once?redirectedfrom=MSDN&view=msvc-170

"Failure with `#include once` can occur in complex projects when file system aliasing or aliased include paths prevent the compiler from identifying identical include files by canonical path."

0 Reply

Krishnakumar
October 11, 2023 4:15 am

Some pragmas are standardised separately to the C++ standard. For example, the OpenMP pragmas that are predominant in multi-threaded shared-memory scientific computing is standarding by the OpenMP consortium. Different compilers support different versions of the openMP standard by including the toolchain-supplied openmp header in the user code and by just invoking a compiler flag.

For example, with a standard gcc/libc installation, one can simply do `#include <omp.h>` and start using pragmas within the source code. The program can be compiled as an openmp-threaded program using the `-fopenmp` flag, or can be compiled as a standard serial program by omitting it. No additional installation/configuration is needed.

0 Reply

IceFloe
Reply to Krishnakumar 21 February 21, 2024 4:23 am

Thank you for the additional information, it was interesting to read

0 Reply

tom
October 6, 2023 11:46 am

In the section **Header guards do not prevent a header from being included once into different code files**, in the modified code, why do we need to `#include "square.h"` in the `square.cpp` file? I think that when we `#include` that `square.h` in the `main.cpp`, it will import the forward declarations and connect that to the definition in the `square.cpp` file anyway. Please let me know, thank you!

0 Reply

MOEGMA25

Reply to tom[22] November 25, 2023 5:00 am

If there are any errors the compiler will complain first instead of the linker, this will save you time.

"This allows the compiler to catch certain kinds of errors at compile time instead of link time."

0 Reply

Alex Author

Reply to tom[22] October 6, 2023 11:53 am

It's a best practice for a .cpp file to always include it's paired header.

0 Reply

Edward

October 5, 2023 4:46 pm

So from my understanding `#pragma once` is not standardized but most developers are using it anyway?

What about modules? Will modules be the best of both worlds (easier to write + standardized)?

0 Reply

Alex Author

Reply to Edward[23] October 5, 2023 9:55 pm

> So from my understanding #pragma once is not standardized but most developers are using it anyway?

Correct (I don't know if I'd say most, but certainly a non-trivial percentage).

> What about modules? Will modules be the best of both worlds (easier to write + standardized)?

Modules are standardized. I'm not yet familiar enough with the intricacies of writing them to say whether they are easier to write than headers. They will definitely be easier to use.

1 Reply

Krishnakumar

Reply to Alex[24] November 16, 2023 4:25 am

Waiting for modules. Has been a long wait. Will have to wait even longer.

0 Reply

Nathaniel Ramirez
October 1, 2023 3:11 pm

I need to reread this a few more times to see what I am missing.

The big thing for me is that I don't understand why two files are both able to #include <iostream> without any issue, I would assume it would be the same thing as trying to include a header into two files. Does it act like a forward definition in the second file?

Last edited 6 months ago by Nathaniel Ramirez

0 Reply

teresuki
Reply to Nathaniel Ramirez [25] October 2, 2023 2:55 am

Let's first assume we have two distinct cpp files (add.cpp and main.cpp) and they do not include each other.

We can simply use #include <iostream> in both of these files, because #include directive here simply "copies" the content of the iostream header into the place you use i (just like you can paste the same paragraph into two different word files).

Now, in another case, let's say we have a header file add.h which has #include <iostream>. Then, in our main.cpp we include both add.h and iostream:

```
#include "add.h"
#include <iostream>
```

Since in add.h we already include iostream, in main we are essentially including iostream twice (one from #include "add.h" and one from #include <iostream>). But no error will occurred, because iostream has header guard, so essentially the second iostream in main.cpp will be ignored.

3 Reply

Nathaniel Ramirez
Reply to teresuki [26] October 2, 2023 7:42 am

Two questions:

1. Both files will receive the contents of the iostream header, as the macro is no longer defined once the first file is preprocessed. How does the second file not trigger the "no second definitions" rule? Is it just that <iostream> and all of its includes do not contain definitions?

2. This is more of a mechanism question: main does not receive IOSTREAM because of the header guard, what makes it able to compile? Does the include act as a forward definition?

I need to do more research on how the preprocessor works.

0 Reply

teresuki

Reply to Nathaniel Ramirez 27 October 2, 2023 8:18 am

There might be some misunderstanding so I will repeat my previous comment a little bit. I will give example of two cases here:

1. main.cpp does not include add.h:

Your add.h will look like this:

```
1 | #include <iostream>
2 | // Other code below
```

Your main.cpp will look like this:

```
1 | #include <iostream>
2 | // Other code below.
```

Regarding your first question: There's only one definition for, for example: std::cout inside iostream. We did not redefine std::cout elsewhere.

2. main.cpp includes both add.h and iostream. And add.h also include iostream:

add.h is still the same:

```
1 | #include <iostream>
2 | // Other code below
```

Now, our main.cpp also include add.h as well:

```
1 | #include "add.h"
2 | #include <iostream>
3 | // Other code below
```

Regarding your second question. main.cpp does not include iostream (at line 2) again thanks to the header guard (iostream is included in add.h already, so #ifndef will be "false", and the code inside the #ifndef and #endif will be ignored.), that's why main.cpp will compile normally.

Last edited 6 months ago by teresuki

0 Reply

Nathaniel Ramirez
Reply to teresuki 28 October 2, 2023 8:31 am

1. That makes a lot more sense, so the no second definition file only applies per object file?
2. My question here is a conceptual one, main.cpp does not include iostream because of the header guard. But the include is necessary if you want to use std::cout in main.cpp. How is it that despite main.cpp including nothing (i.e. iostream is already defined and #ifndef is false), it is able to use std::cout?

Last edited 6 months ago by Nathaniel Ramirez

0 Reply

teresuki
Reply to Nathaniel Ramirez 29 October 2, 2023 9:00 am

1. The rule applies per file and per program too (lesson 2.7). You might be confused about the per program part (since we are including iostream two times in two different files). But, we are including the same header file (the same content) and there is only one single definition of, let's say, std::cout in the entire program. EDIT: My previous comment is misleading. I am revising it.
2. Theoretically you could remove the #include <iostream> inside main.cpp and you can still use std::cout inside main.cpp, because you already have iostream in #include "add.h". I think this is mentioned in lesson 2.11 (transitive include). But I don't recommend doing it, since at any point if I don't include iostream in add.h anymore, main.cpp cannot use std::cout anymore.

Last edited 6 months ago by teresuki

0 Reply

Nathaniel Ramirez
Reply to teresuki 30 October 2, 2023 9:08 am

"But, we are including the same header file (the same content) and there is only one single definition of, let's say, std::cout in the entire program."

So including iostream twice does not redefine std::cout?

0 Reply

teresuki

Reply to Nathaniel Ramirez [31]
October 2, 2023 9:12 am

Yes, that's correct.

0 Reply

Nathaniel Ramirez

Reply to teresuki [32] October 2, 2023 9:15 am

I think that is where my confusion lied. I assumed including iostream multiple times would also define std::cout multiple times but there must be more advanced guards...

0 Reply

Alex Author

Reply to Nathaniel Ramirez [33]
October 2, 2023 1:26 pm

`std::cout` is not defined in the `iostream` header. It is forward declared in the `iostream` header so it can be used wherever `iostream` is #included.

Where it is actually defined is implementation-defined.

3 Reply

Nathaniel Ramirez

Reply to Alex [34]
October 2, 2023 2:34 pm

Thank you. I was trying to look at the iostream.h file in Visual Studio and could not identify the forward declaration so that was the missing piece of the puzzle for me.

Last edited 6 months ago by Nathaniel Ramirez

0 Reply

Krishnakumar
August 29, 2023 4:17 am

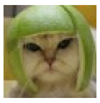
><PROJECT><PATH><FILE>H , <FILE><LARGE RANDOM NUMBER>H

Missing a `_` before `H`?

I am not sure if hard-coding the file's path relative to the project's root in a header guard name is reasonable, since the relative paths might change.

i.e. a situation akin to that mentioned in the include directories best practices section earlier from this tutorial series.

0 Reply



Alex Author
Reply to Krishnakumar [35] August 29, 2023 2:25 pm

Fixed the missing underscores, thanks for noticing.

Boost uses PROJECT_PATH_FILE_EXTENSION. If it's good enough for them...

0 Reply

Krishnakumar
Reply to Alex [36] October 11, 2023 4:08 am

Ok. IMHO, relatively mature boost sub-projects rarely change directory structure (and hence the relative path can be used for header guards).

Thanks a lot for considering the minor suggestion of missing underscores. Happy to move on.

0 Reply

Krishnakumar
August 29, 2023 4:07 am

Thank you for this excellent lesson, Alex!

Last edited 7 months ago by Krishnakumar

0 Reply

vstar
August 7, 2023 8:47 pm

I have a question, if my function return type is not sure, I define it as auto, the compiler reports an error saying that it must be defined, then why do I still write an extra code file, just write it in the header file, right?

0 Reply

> **Alex** Author
> Reply to vstar [37] August 11, 2023 5:11 pm
>
> Functions need to have a specific return type. Using `auto` as the return type just asks the compiler to deduce the return type from the body of the function. We cover this in a later lesson.
>
> 1 Reply

Emeka Daniel
August 3, 2023 10:35 am

I use both traditional and modern header guards, the traditional is first. Basically just double proofing my code, because there was an instance that traditional header guards failed for me, it had to to with the unconventional way I was using transitive headers...traditional header guards just couldn't keep up.

Anyway I stopped such coding behaviour and now double proof my code just in case I find a way to break traditional header guards again.

0 Reply

> **Krishnakumar**
> Reply to Emeka Daniel [38] August 29, 2023 4:49 am
>
> Can you please provide a minimal reproducible example wherein traditional header guards failed. I am curious to see where they break.

0 Reply

Emeka Daniel

Reply to Krishnakumar[39] August 29, 2023 5:59 pm

I am sorry :(, I have no recollection of the exact order in which I included my files to make it fail, I just know that I dependend alot on including header files transitivly by putting all of my related headers in one file and scattering it about.
I think I made a comment about it back then, can't remember the particular chapter though.

But I do remember the outcome of the failure. My compiler was telling that I had include twice of each header specific content in what was back then my miscellany.cpp and blackjack.cpp files, what was weird though was that if I removed a particular header file which was functions.h -it was among the files that i used to transitivly include other header files- from anyone of the files mentioned above, compilation would work without error. I thought maybe the compiler had broken or something, till I then used the #pragma once directive and it all worked even with my functions.h included everywhere. Then I knew it was the traditional header guards that had failed.

If I had known I would have documented the outcome and cause so as to dictate weaknesses and possible preventive measures for the problem, but I was new to programming then and I just carried on after the whole incident.

0 Reply

Krishnakumar

Reply to Emeka Daniel[40] October 11, 2023 5:04 am

Alright. Thanks!

0 Reply

Sphar

July 26, 2023 10:31 pm

```
Now when the program is compiled, function getSquareSides will have just
one definition (via square.cpp), so the linker is happy. File main.cpp is
able to call this function (even though it lives in square.cpp) because
it includes square.h, which has a forward declaration for the function
(the linker will connect the call to getSquareSides from main.cpp to the
definition of getSquareSides in square.cpp).
```

So in this note, is it safe to say that it's alright to have each translation unit (source files) to have a copy of the forward declaration from the header file as each translation unit are not aware of each other's existence. The purpose of duplicate forward declarations in each translation unit is for them to be aware of the existence of the function which is defined in square.cpp and its momentary usage in main.cpp. =

0 Reply

Alex Author

Reply to Sphar 41 July 29, 2023 2:57 pm

Yes.

1 Reply

Ther

July 14, 2023 6:42 am

There is one thing I do not understand. Under the chapter **Updating our previous example with header guards** three files are presented. Now, let's say that they are processed in following order:

1. **square.h**,
2. **wave.h**,
3. **main.cpp**.

square.h gets processed to:

```
int getSquareSides()
{
    return 4;
}
```

wave.h also gets processed to (it is another file hence SQUARE_H is not defined):

```
int getSquareSides()
{
    return 4;
}
```

Now when **main.cpp** is being processed how come it doesn't copy processed content of **square.h** and **wave.h** which would result in multiple definitions of getSquareSides() function?

0 Reply

Keith Wallace

Reply to Ther 42 July 14, 2023 10:43 am

You say "wave.h also gets processed to **(it is another file hence SQUARE_H is not defined)**..."

I don't think this is the case. I read this as each **.cpp** code file has its own set of `SQUARE_H` and `WAVE_H`, rather than each **.h** file does.

So, as `main.cpp` is being built by the precompiler, it essentially copy/pastes the entire `square.h` file (including its header guard) and the entire `wave.h` file (including its header guard) into itself, then starts working through it setting and checking for SQUARE_H and WAVE_H definitions **within that one file**.

2 Reply

Alex Author

Reply to Keith Wallace 43 July 14, 2023 1:25 pm

Confirming this is correct. :)

Only source (.cpp) files get compiled directly. Header (.h) files get copied into .cpp files before they are processed.

6 Reply

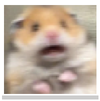
Ther

Reply to Keith Wallace 43 July 14, 2023 11:54 am

Okay, this actually makes total sense. I thought that that process happens both for **.cpp** and **.h** files. Thank you for your answer.

Last edited 9 months ago by Ther

2 Reply



Zoltan

June 19, 2023 5:58 am

I get the whole thing, really I think I understand everything EXCEPT: how does the header file know that the definition that I'm referencing it to is in the correct .cpp file?

e.g. if I'm developing a game and I have an enemy.h and enemy.cpp, and then I have a player.h and player.cpp, how does the function "void attack();" in player.h know that I'm referring it to "void attack(){...}" in player.cpp?

Is it the identical name? Is it the identical function name with the same amount and type of function arguments?

0 Reply

Nadir

Reply to Zoltan 44 August 13, 2023 4:39 am

It doesn't. If you define void attack(){...} in both player.cpp and enemy.cpp you're violating the ODR. The linker will complain that it found two definitions of void attack() and can't resolve the ambiguity.

If void attack() is only defined in player.cpp (probably what you meant), the linker will find it. Header file doesn't need to "know" where the definition is.

0 Reply

Ilya Chalov
Reply to Zoltan 44 June 19, 2023 1:05 pm

> how does the header file know that the definition that I'm referencing it to is in the correct .cpp file?

The file can't know anything, since it's just data. A program like a compiler might know something.

> how does the function "void attack();" in player.h know that I'm referring it to "void attack(){...}" in player.cpp?

The function doesn't know anything. When compiling one file, the compiler does not remember what it saw in other files. The linker links the function call and its definition.

`void attack();` in the file `player.h` is a forward declaration that tells the compiler that this function exists somewhere.

`void attack(){...}` in the file `player.cpp` is the function definition.

> Is it the identical name? Is it the identical function name with the same amount and type of function arguments?

Yes, the compiler distinguishes functions by their names, the number of parameters and the type of parameters. If the functions have the same names but different parameters, then these are different functions.

(I hope I have not made a mistake anywhere in these explanations. If I made a mistake, I hope I will be corrected.)

0 Reply

Krishnakumar
May 28, 2023 10:38 am

For those interested, there is a nifty little tool called `guardonce` https://github.com/cgmb/guardonce that can convert all the header files in the project to switch bac & forth between header guards style and pragma once style :)

Last edited 10 months ago by Krishnakumar

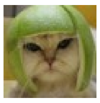
1 Reply

Krishnakumar
May 28, 2023 10:30 am

>These **user-defined types** are typically defined in header files

Program-defined types is more technically correct, perhaps?

0 Reply



Alex Author
Reply to Krishnakumar 45 May 31, 2023 2:26 pm

Yeah, but then I have to define what "program-defined" means, and I don't want to have that discussion this early. Instead, I've changed it to "custom types", which will suffice for now.

1 Reply

Krishnakumar
Reply to Alex 46 October 11, 2023 4:09 am

Thanks Alex. I think I noticed one or two uses of user-defined types in the lesson/comments thus far. But not super important anyway.

0 Reply

Krishnakumar
May 28, 2023 9:58 am

In `main.cpp` (multipl examples), the lines:

```
#include "square.h"
#include "geometry.h"
```

violate the sort "alphabetically within each category/block" summarised at the end of the previous lesson (in the section on header file ordering).

This should be sorted as:

```
#include "geometry.h"
#include "square.h"
```

I do understand that this calls for significant revision in the subsequent explanatory text, but will aid with consistency of following the best practices we recommend. Sticking to current order impacts especially those who have implemented the automatic sorting of header files to adhere to your prescribed ordering hierarchy (e.g. through customisation of `clang-format` rules).

Another quicker alternative would be to just note that these examples do not intentionally follow the sort order best practices advocated earlier for pedagogical purposes (the lesson already adopts this style of noting exceptions from recommended practice e.g. it states upfront that we intentionally violate the best practice guide of not defining functions within header files purely for teaching purposes).

Last edited 10 months ago by Krishnakumar

0 Reply

Alex Author
Reply to Krishnakumar [47] May 31, 2023 2:24 pm

Since geometry.h is basically empty, I just renamed it to wave.h and updated the names. Much faster than reordering everything.

1 Reply

Krishnakumar
Reply to Alex [48] October 11, 2023 4:10 am

Yes. Thank you, Alex.

0 Reply

yateen
May 20, 2023 12:37 pm

is int x a declaration or a definition? in 2.7 you mentioned it was a declaration, so why does it violate the one definition rule? if it is not, is there any way to have variable declaration?

0 Reply

Alex Author
Reply to yateen [49] May 22, 2023 3:25 pm

It's both a definition and a declaration. All definitions are declarations.

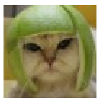
We cover variable forward declarations in lesson https://www.learncpp.com/cpp-tutorial/external-linkage-and-variable-forward-declarations/

0 Reply

yateen
Reply to Alex [50] May 24, 2023 12:40 am

is there any way to declare a variable or is it only possible for a function?

0 Reply



Alex Author

Reply to yateen 51 May 24, 2023 1:25 pm

Yes, it's possible to declare a variable. It's covered in the lesson I linked in the prior comment.

0 Reply

yateen

Reply to Alex 52 May 26, 2023 3:49 am

oh alright, thank you.

0 Reply

J34NP3T3R

May 11, 2023 7:23 pm

Question with this note

```
For advanced readers

In large programs, it’s possible to have two separate header files (included
from different directories) that end up having the same filename (e.g.
directoryA\config.h and directoryB\config.h). If only the filename is used
for the include guard (e.g. CONFIG_H), these two files may end up using the
same guard name. If that happens, any file that includes (directly or
indirectly) both config.h files will not receive the contents of the include
file to be included second. This will probably cause a compilation error.

Because of this possibility for guard name conflicts, many developers
recommend using a more complex/unique name in your header guards. Some good
suggestions are a naming convention of <PROJECT><PATH><FILE>H , <FILE><LARGE
RANDOM NUMBER>H, or <FILE><CREATION DATE>_H
```

using a naming convention solution for header guards would resolve this issue since the preprocessor is now checking for a unique name, but my confusion is in the fact that the file names are still the same "config.h" so when you #include "config.h" wouldn't that still generate an error of some sort ?

and How does #pragma once work ? does it check using the file name ?
like if its used in math.h does pragma once check if it already included contents from a file named "math.h" ?
so like the example above, if two files have the same file name but different contents then the contents of the second math.h to be #included will be discarded ?

Last edited 11 months ago by J34NP3T3R

0 Reply

Alex Author

Reply to J34NP3T3R [53] May 14, 2023 5:07 pm

If you just `#include "config.h"`, if there are multiple config.h files in the include path then it's implementation specific which one you'll get. That's why such files are often left inside subdirectories, so you can `#include "somedir1\config.h"` or `#include "somedir2\config.h"` and know you'll get the right one.

#pragma is implementation specific as well, so it works however the compiler decides it should work.

1 Reply

J34NP3T3R

Reply to Alex [54] May 15, 2023 5:15 am

thanks

0 Reply

Rescued

May 1, 2023 10:07 am

Were you tired while writing this lesson? why did you declare functions in a file and wrote functions in a header?

0 Reply

Alex Author

Reply to Rescued [55] May 2, 2023 10:30 pm

No, it's just the easiest way to demonstrate the concept of header guards using functionality we've already covered (since we haven't covered global variables, program-defined types, or templates yet).

1 Reply

Krishnakumar

Reply to Alex [56] August 29, 2023 4:44 am

custom types.

0 Reply

Timo

April 28, 2023 12:52 pm

For me, including square.h and geometry.h inside main.cpp does compile. Can you tell me why it does compile?

Last edited 11 months ago by Timo

0 Reply

Alex Author

Reply to Timo [57] May 2, 2023 3:11 pm

It will compile as long as square.h has header guards. That way when the contents of square.h are included by both main.cpp and geometry.h, the second inclusion will be excluded by the header guards and compiled out.

0 Reply

Links

1. https://www.learncpp.com/author/Alex/
2. https://www.learncpp.com/cpp-tutorial/forward-declarations/
3. javascript:void(0)
4. https://www.learncpp.com/cpp-tutorial/how-to-design-your-first-programs/
5. https://www.learncpp.com/
6. https://www.learncpp.com/cpp-tutorial/header-files/
7. https://www.learncpp.com/header-guards/
8. https://www.learncpp.com/cpp-tutorial/chapter-14-summary-and-quiz/
9. https://www.learncpp.com/cpp-tutorial/scope-duration-and-linkage-summary/
10. https://gravatar.com/
11. https://www.learncpp.com/cpp-tutorial/header-guards/#comment-595989
12. https://www.learncpp.com/cpp-tutorial/header-guards/#comment-596034
13. https://www.learncpp.com/cpp-tutorial/header-guards/#comment-596039
14. https://www.learncpp.com/cpp-tutorial/header-guards/#comment-595352
15. https://www.learncpp.com/cpp-tutorial/header-guards/#comment-594476
16. https://www.learncpp.com/cpp-tutorial/header-guards/#comment-594201
17. https://www.learncpp.com/cpp-tutorial/header-guards/#comment-594173
18. https://www.learncpp.com/cpp-tutorial/header-guards/#comment-594154
19. https://www.learncpp.com/cpp-tutorial/header-guards/#comment-592126
20. https://www.learncpp.com/cpp-tutorial/header-guards/#comment-590536

21. https://www.learncpp.com/cpp-tutorial/header-guards/#comment-588526
22. https://www.learncpp.com/cpp-tutorial/header-guards/#comment-588319
23. https://www.learncpp.com/cpp-tutorial/header-guards/#comment-588264
24. https://www.learncpp.com/cpp-tutorial/header-guards/#comment-588286
25. https://www.learncpp.com/cpp-tutorial/header-guards/#comment-588026
26. https://www.learncpp.com/cpp-tutorial/header-guards/#comment-588035
27. https://www.learncpp.com/cpp-tutorial/header-guards/#comment-588073
28. https://www.learncpp.com/cpp-tutorial/header-guards/#comment-588075
29. https://www.learncpp.com/cpp-tutorial/header-guards/#comment-588077
30. https://www.learncpp.com/cpp-tutorial/header-guards/#comment-588081
31. https://www.learncpp.com/cpp-tutorial/header-guards/#comment-588082
32. https://www.learncpp.com/cpp-tutorial/header-guards/#comment-588083
33. https://www.learncpp.com/cpp-tutorial/header-guards/#comment-588084
34. https://www.learncpp.com/cpp-tutorial/header-guards/#comment-588117
35. https://www.learncpp.com/cpp-tutorial/header-guards/#comment-586315
36. https://www.learncpp.com/cpp-tutorial/header-guards/#comment-586386
37. https://www.learncpp.com/cpp-tutorial/header-guards/#comment-585256
38. https://www.learncpp.com/cpp-tutorial/header-guards/#comment-585041
39. https://www.learncpp.com/cpp-tutorial/header-guards/#comment-586324
40. https://www.learncpp.com/cpp-tutorial/header-guards/#comment-586390
41. https://www.learncpp.com/cpp-tutorial/header-guards/#comment-584610
42. https://www.learncpp.com/cpp-tutorial/header-guards/#comment-583789
43. https://www.learncpp.com/cpp-tutorial/header-guards/#comment-583793
44. https://www.learncpp.com/cpp-tutorial/header-guards/#comment-582140
45. https://www.learncpp.com/cpp-tutorial/header-guards/#comment-580906
46. https://www.learncpp.com/cpp-tutorial/header-guards/#comment-581045
47. https://www.learncpp.com/cpp-tutorial/header-guards/#comment-580904
48. https://www.learncpp.com/cpp-tutorial/header-guards/#comment-581044
49. https://www.learncpp.com/cpp-tutorial/header-guards/#comment-580580
50. https://www.learncpp.com/cpp-tutorial/header-guards/#comment-580665
51. https://www.learncpp.com/cpp-tutorial/header-guards/#comment-580731
52. https://www.learncpp.com/cpp-tutorial/header-guards/#comment-580759
53. https://www.learncpp.com/cpp-tutorial/header-guards/#comment-580314
54. https://www.learncpp.com/cpp-tutorial/header-guards/#comment-580368
55. https://www.learncpp.com/cpp-tutorial/header-guards/#comment-579944
56. https://www.learncpp.com/cpp-tutorial/header-guards/#comment-580008
57. https://www.learncpp.com/cpp-tutorial/header-guards/#comment-579892